



# PROGRAMACIÓN II - UNIDAD 2

# Agenda

---

1. Expresiones lambda.
2. Arrays.
3. Número aleatorio

# 1 - Expresiones Lambda

- Una función lambda es una pequeña función anónima.
- Una función lambda puede tomar cualquier número de argumentos.

# 1 - Expresiones Lambda

## *Sintaxis*

*lambda arguments : expression*

Se ejecuta la expresión y se devuelve el resultado.

```
x = lambda a: a + 10  
print(x(5))
```

15

```
x = lambda a, b: a * b  
print(x(5, 6))
```

30

```
x = lambda a, b, c: a + b + c  
print(x(5, 6, 2))
```

13

# 1 - Expresiones Lambda

¿Por qué utilizar las funciones Lambda?

El poder de lambda se muestra mejor cuando se usa como una función anónima dentro de otra función.

Supongamos que tiene una definición de función que toma un argumento, y ese argumento se multiplicará por un número desconocido.

# 1 - Expresiones Lambda

Asumamos esta función:

```
def myfunc(n):  
    return lambda a : a * n
```

Podemos usarla para duplicar cualquier valor enviado:

```
def myfunc(n):  
    return lambda a : a * n  
  
mydoubler = myfunc(2)  
  
print(mydoubler(11))
```

22

También podemos usarla para triplicar el valor enviado:

```
def myfunc(n):  
    return lambda a : a * n  
  
mytripler = myfunc(3)  
  
print(mytripler(11))
```

33

## 2 - Arrays

Python no maneja arrays de manera nativa. Podemos emularlos usando listas, armando listas que contengan listas.

También podemos incluir la librería NumPy que da soporte para numerosas funciones matemáticas incluyendo el manejo de arrays.

<https://numpy.org/>



# 2 - Arrays



El paquete fundamental para la informática científica con Python

EMPEZAR

**NumPy v1.19.0** Primera versión única de Python 3: interfaz Cython para completar `numpy.random`



## POTENTES MATRICES N-DIMENSIONALES

Rápidos y versátiles, los conceptos de vectorización, indexación y transmisión de NumPy son los estándares de facto de la computación de arreglo en la actualidad.

## HERRAMIENTAS DE COMPUTACIÓN NUMÉRICA

NumPy ofrece funciones matemáticas completas, generadores de números aleatorios, rutinas de álgebra lineal, transformadas de Fourier y más.

## INTEROPERABLE

NumPy admite una amplia gama de hardware y plataformas informáticas, y funciona bien con bibliotecas distribuidas, GPU y de arreglos dispersos.

## PERFORMANTE

El núcleo de NumPy es un código C bien optimizado. Disfrute de la flexibilidad de Python con la velocidad del código compilado.

## FÁCIL DE USAR

La sintaxis de alto nivel de NumPy lo hace accesible y productivo para programadores de cualquier origen o nivel de experiencia.

## FUENTE ABIERTA

Distribuido bajo una [licencia BSD](#) liberal, NumPy es desarrollado y mantenido públicamente en [GitHub](#) por una [comunidad](#) vibrante, receptiva y diversa.




## 2 - Arrays

```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])

print(arr)
```



```
[1 2 3 4 5]
```

Si deseamos utilizar NumPy con un nombre reducido:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```



```
[1 2 3 4 5]
```

## 2 - Arrays

### Comprobación de la versión de NumPy

La cadena de versión se almacena en el atributo `__version__`.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)

print("")

print(np.__version__)
```



C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37\_64\python.exe

```
[1 2 3 4 5]
```

```
1.19.1
```

```
Press any key to continue . . .
```

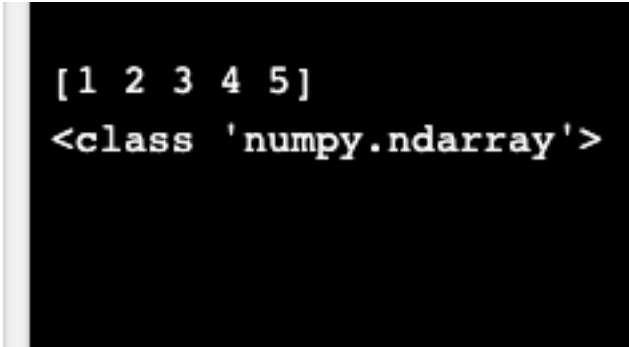
## 2 - Arrays

¿Qué tipo poseen los arrays creados por medio de NumPy?

Son de tipo ndarray

Podemos crear un objeto ndarray NumPy usando la función `array()`.

```
import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5])  
  
print(arr)  
  
print(type(arr))
```



```
[1 2 3 4 5]  
<class 'numpy.ndarray'>
```

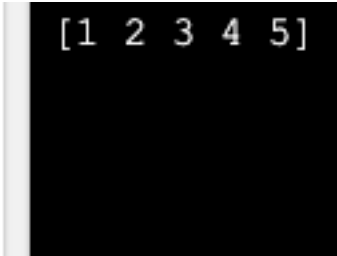
## 2 - Arrays

Para crear un `ndarray`, podemos pasar una lista, tupla o cualquier objeto similar a una matriz en el método `array()`, y se convertirá en un `ndarray`:

```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```

A terminal window with a black background and a light gray vertical scrollbar on the left. The text "[1 2 3 4 5]" is displayed in a light blue monospace font at the top of the window.

```
[1 2 3 4 5]
```

## 2 - Arrays

### Dimensiones en matrices

Una dimensión en matrices es un nivel de profundidad de la matriz (matrices anidadas).

**Matriz anidada:** son matrices que tienen matrices como elementos.

### Matrices 0-D

Las matrices 0-D, o escalares, son los elementos de una matriz. Cada valor de una matriz es una matriz 0-D.

```
import numpy as np  
  
arr = np.array(42)  
  
print(arr)
```

42

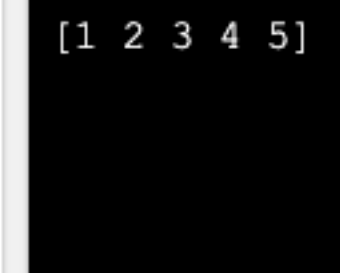
# 2 - Arrays

## Matrices 1-D

Una matriz que tiene matrices 0-D como sus elementos se denomina matriz unidimensional o 1-D.

Estos son los arreglos más comunes y básicos.

```
import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5])  
  
print(arr)
```



```
[1 2 3 4 5]
```

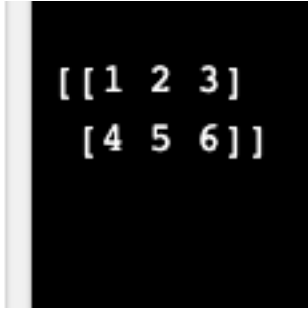
## 2 - Arrays

### Matrices 2-D

Una matriz que tiene matrices 1-D como sus elementos se denomina matriz 2-D.

NumPy tiene un submódulo completo dedicado a operaciones matriciales llamado `numpy.mat`

```
import numpy as np  
  
arr = np.array([[1, 2, 3], [4, 5, 6]])  
  
print(arr)  
|
```



```
[[1 2 3]  
 [4 5 6]]
```

## 2 - Arrays

### Matrices 3-D

Una matriz que tiene matrices 2-D (matrices) como sus elementos se llama matriz 3-D.

```
import numpy as np  
  
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])  
  
print(arr)
```

```
[[[1 2 3]  
  [4 5 6]]  
  
 [[1 2 3]  
  [4 5 6]]]
```



## 2 - Arrays

**Para comprobar el número de dimensiones** de un Array NumPy proporciona el atributo **ndim** que devuelve un número entero que nos dice cuántas dimensiones tiene la matriz.

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

0  
1  
2  
3

# 2 - Arrays

## Indexación de matrices NumPy

### Acceso a elementos de matriz

La indexación de matrices es lo mismo que acceder a un elemento de la matriz.

Puede acceder a un elemento de matriz consultando su número índice.

Los índices en las matrices NumPy comienzan con 0, lo que significa que el primer elemento tiene índice 0 y el segundo tiene índice 1, etc.

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[0])
```

1

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[2] + arr[3])
```

7

## 2 - Arrays

Para acceder a elementos de matrices 2-D, podemos usar enteros separados por comas que representan la dimensión y el índice del elemento.

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st dim: ', arr[0, 1])
```

```
2nd element on 1st dim:  2
```

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('5to elemento de la segunda dimensión: ',
      arr[1, 4])
```

```
5to elemento de la segunda dimensión:  10
```

## 2 - Arrays

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print(arr[0, 1, 2])
```

6

Indexación negativa:

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('Last element from 2nd dim: ', arr[1, -1])
```

Last element from 2nd dim: 10

## 2 - Arrays

### ***Corte de Matrices***

Cortar en python significa tomar elementos de un índice dado a otro índice dado.

Se utiliza pasando el índice de entrada y salida: `[start:end]`

También podemos pasarle un salto o paso (step): `[start:end:step]`

Si no pasamos el inicio se considera 0


Si no pasamos end la longitud total del array es tomado en consideración.

Si no pasamos el paso se considera 1

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```



[2 3 4 5]

## 2 - Arrays

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[4:])
```

```
[5 6 7]
```

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[:4])
```

```
[1 2 3 4]
```

## 2 - Arrays

Índice negativo


```
import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[-3:-1])
```



[5 6]

Utilizando step

```
import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[1:5:2])
```

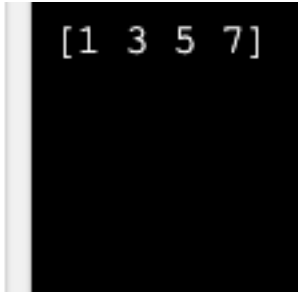


[2 4]

## 2 - Arrays

Desde el comienzo hasta el final con un step de 2

```
import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[::2])
```



```
[1 3 5 7]
```

Recorte de elementos en un array 2D

```
import numpy as np  
  
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  
  
print(arr[1, 1:4])
```



```
[7 8 9]
```



## 2 - Arrays

De cada elemento del array retorna el que se encuentra en el índice 2

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 2])
```

```
[3 8]
```

Retorno de una matriz 2D conteniendo los elementos del 1 al 4 excluyendo el 4 de ambos elementos

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 1:4])
```

```
[[2 3 4]
 [7 8 9]]
```

## 2 - Arrays

### *Diferencia entre Copy y View*

La principal diferencia entre una copia y una vista de una matriz es que la copia crea una nueva matriz y la vista es solo una vista de la matriz original.

La copia es *propietaria* de los datos y cualquier cambio realizado en la copia **no afectará** la matriz original, y cualquier cambio realizado en la matriz original **no afectará** la copia.

La vista *no es propietaria* de los datos y cualquier cambio realizado en la vista **afectará** la matriz original, y cualquier cambio realizado en la matriz original **afectará** la vista.

## 2 - Arrays

copy()

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42

print(arr)
print(x)
```

```
[42  2  3  4  5]
[1  2  3  4  5]
```

view()

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42

print(arr)
print(x)
```

```
[42  2  3  4  5]
[42  2  3  4  5]
```

## 2 - Arrays

### *Forma de una matriz*

La forma de una matriz es el número de elementos en cada dimensión.

¿Cómo obtengo la forma de una matriz?

Las matrices NumPy tienen un atributo llamado **shape** que devuelve una tupla con el número de elementos correspondientes a cada índice.

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(arr.shape)
```



```
(2, 4)
```

## 2 - Arrays

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('shape of array :', arr.shape)
```

```
[[[[[1 2 3 4]]]]]
shape of array : (1, 1, 1, 1, 4)
```

## 2 - Arrays

### *Cambiando la forma de las matrices (dimensiones)*

Reformar significa cambiar la forma de una matriz.

La forma de una matriz es el número de elementos en cada dimensión.

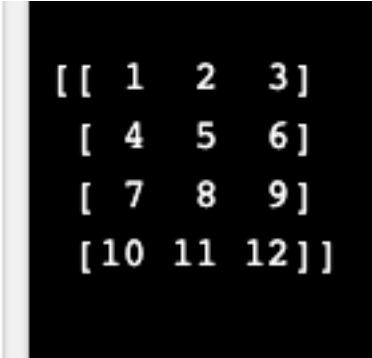
Al remodelar, podemos agregar o eliminar dimensiones o cambiar el número de elementos en cada dimensión. La función `reshape()` retorna una vista.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```



```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

## 2 - Arrays

```
import numpy as np  
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  
newarr = arr.reshape(2, 3, 2)  
print(newarr)
```

```
[[[ 1  2]  
  [ 3  4]  
  [ 5  6]]  
  
 [[ 7  8]  
  [ 9 10]  
 [11 12]]]
```

## 2 - Arrays

*¿Podemos reformarnos en cualquier forma?*

Sí, siempre que los elementos necesarios para remodelarla sean iguales en ambas formas.

Podemos remodelar una matriz 1D de 8 elementos en 4 elementos en una matriz 2D de 2 filas, pero no podemos remodelarla en una matriz 2D de 3 elementos y 3 filas, ya que eso requeriría  $3 \times 3 = 9$  elementos.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(3, 3)
print(newarr)
```

```
Traceback (most recent call last):
  File "demo_numpy_array_reshape_error.py", line 5, in <module>
    ValueError: cannot reshape array of size 8 into shape (3,3)
```



## 2 - Arrays

### *Dimensión desconocida*

Se permite tener una dimensión "desconocida".

Lo que significa que no tiene que especificar un número exacto para una de las dimensiones en el método de cambio de forma.

Pase **-1** como valor y NumPy calculará este número por usted.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

newarr = arr.reshape(2, 2, -1)

print(newarr)
```

```
[[[1 2]
  [3 4]

  [5 6]
  [7 8]]]
```

## 2 - Arrays

### *Aplanando las matrices*

Aplanar una matriz significa convertir una matriz multidimensional en una matriz 1D.

Podemos utilizar `reshape(-1)` para hacer esto.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

newarr = arr.reshape(-1)

print(newarr)
```

A terminal window with a black background and white text. The text shows the output of the Python code: a single row of six numbers, [1 2 3 4 5 6], enclosed in square brackets.

```
[1 2 3 4 5 6]
```

## 2 - Arrays

### *Iterando matrices*

Iterar significa pasar por los elementos uno por uno.

Cuando tratamos con matrices multidimensionales en NumPy, podemos utilizar el bucle **for** básico de Python.

Si iteramos en una matriz 1-D, pasará por cada elemento uno por uno.

```
import numpy as np

arr = np.array([1, 2, 3])

for x in arr:
    print(x)
```



1  
2  
3

## 2 - Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
    print(x)
```

```
[1 2 3]
[4 5 6]
```

Para iterar cada elemento individualmente como es 2D utilizamos dos for

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
    for y in x:
        print(y)
```

```
1
2
3
4
5
6
```

## 2 - Arrays

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
    print("x represents the 2-D array:")
    print(x)
```

```
x represents the 2-D array:
[[1 2 3]
 [4 5 6]]
x represents the 2-D array:
[[ 7  8  9]
 [10 11 12]]
```

Para iterar cada elemento individualmente como es 3D utilizamos tres for

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
    for y in x:
        for z in y:
            print(z)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
```

## 2 - Arrays

### *Iteración enumerada con `ndenumerate()`*

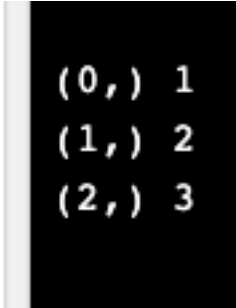
La enumeración significa mencionar el número de secuencia de algunas cosas una por una.

A veces requerimos el índice correspondiente del elemento mientras iteramos, el método `ndenumerate()` se puede usar para esos casos de uso.

```
import numpy as np

arr = np.array([1, 2, 3])

for idx, x in np.ndenumerate(arr):
    print(idx, x)
```



```
(0,) 1
(1,) 2
(2,) 3
```

## 2 - Arrays

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

```
(0, 0) 1
(0, 1) 2
(0, 2) 3
(0, 3) 4
(1, 0) 5
(1, 1) 6
(1, 2) 7
(1, 3) 8
```

# 3 – Número Aleatorio

## ¿Qué es un número aleatorio?

Número aleatorio NO significa un número diferente cada vez. Aleatorio significa algo que no se puede predecir lógicamente.

## Pseudo aleatorio y verdadero aleatorio.

Existe un algoritmo para generar un número aleatorio.

Si hay un programa para generar un número aleatorio, se puede predecir, por lo que no es realmente aleatorio.

Los números aleatorios generados mediante un algoritmo de generación se denominan *pseudoaleatorios*.

## ¿Podemos hacer números verdaderamente aleatorios?

Sí. Para generar un número verdaderamente aleatorio en nuestras computadoras, necesitamos obtener los datos aleatorios de alguna fuente externa. Esta fuente externa son generalmente nuestras pulsaciones de teclas, movimientos del mouse, datos en la red, etc.

No necesitamos números verdaderamente aleatorios, a menos que esté relacionado con la seguridad (por ejemplo, claves de cifrado) o la base de la aplicación sea la aleatoriedad (por ejemplo, ruedas de ruleta digital).



# 3 – Número Aleatorio

Generar un número entero aleatorio de 0 a 100

```
from numpy import random  
  
x = random.randint(100)  
  
print(x)
```

A terminal window with a black background and white text displaying the number 34.

34

Generar aleatoria de un número float

El método rand() del módulo aleatorio devuelve un número float aleatorio entre 0 y 1.

```
from numpy import random  
  
x = random.rand()  
  
print(x)
```

A terminal window with a black background and white text displaying the float value 0.014439623558732184.

0.014439623558732184

# 3 – Número Aleatorio

## *Generar matriz aleatoria*

En NumPy trabajamos con matrices, y puede usar los dos métodos de los ejemplos anteriores para hacer matrices aleatorias.

Enteros

El método `randint()` toma un parámetro `size` en el que puede especificar la forma de una matriz.

```
from numpy import random  
  
x=random.randint(100, size=(5))  
  
print(x)
```

```
[53 58 54 27 24]
```

# 3 – Número Aleatorio

```
from numpy import random  
  
x = random.randint(100, size=(3, 5))  
  
print(x)
```

```
[[90 99 11 30 34]  
 [66 40 63 36 37]  
 [63 35 89 51 58]]
```

*Con números flotantes*

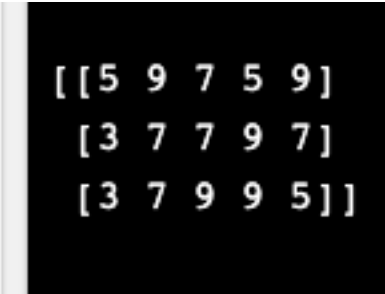
```
from numpy import random  
  
x = random.rand(3, 5)  
  
print(x)
```

```
[[0.03379952 0.78263517 0.9834899 0.47851523 0.02948659]  
 [0.36284007 0.10740884 0.58485016 0.20708396 0.00969559]  
 [0.88232193 0.86068608 0.75548749 0.61233486 0.06325663]]
```

# 3 – Número Aleatorio

El método `choice()` también le permite devolver una *matriz* de valores. Agregue un parámetro `size` para especificar la forma de la matriz.

```
from numpy import random  
  
x = random.choice([3, 5, 7, 9], size=(3, 5))  
  
print(x)
```

A 3x5 matrix of random values displayed in a terminal window. The values are integers ranging from 3 to 9.

```
[[5 9 7 5 9]  
 [3 7 7 9 7]  
 [3 7 9 9 5]]
```